LA-UR-21-25422
Approved for public release; distribution is unlimited.

Title: Searching for Speed With SIMD

Author(s): Mastripolito, Benjamin Philip

Intended for: LANL PCSRI Presentation

Issued: 2021-06-23 (rev.1)

Los Alamos NATIONAL LABORATORY

Searching for Speed With SIMD

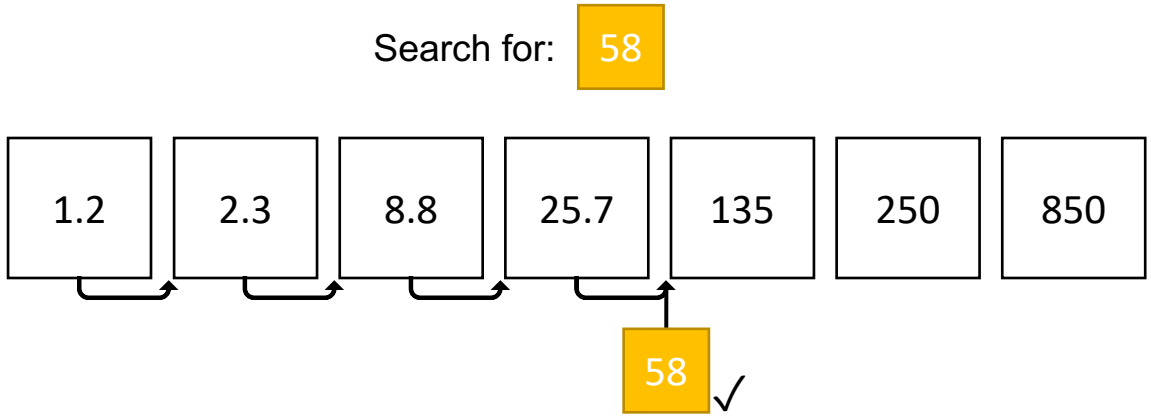
Benjamin Mastripolito

June 15th, 2021

LA-UR-21-25422

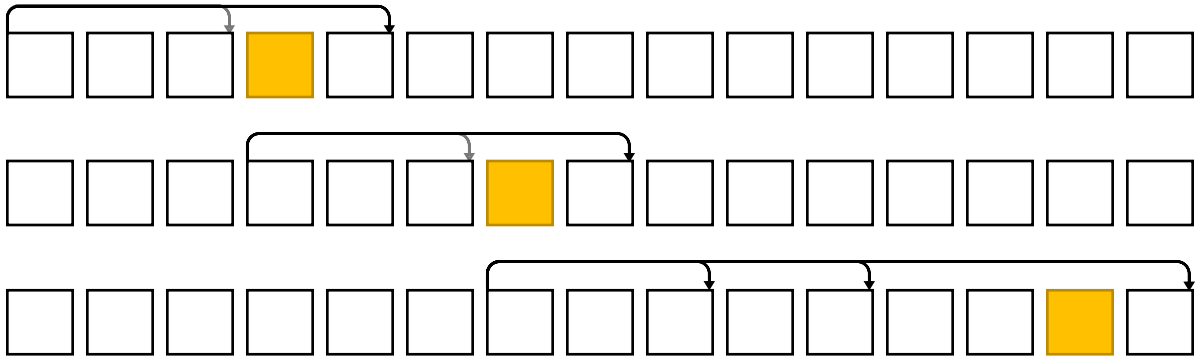
The Problem

- EOSPAC: interpolation on equation-of-state tables (2D arrays)
- Interpolation requires searching for elements bounding target in an array



Hunt & Locate

- The old search algorithm in EOSPAC
- Jumps to produce bounds within which to perform binary search
- Uses previous index to start next search



Constraints 🔒

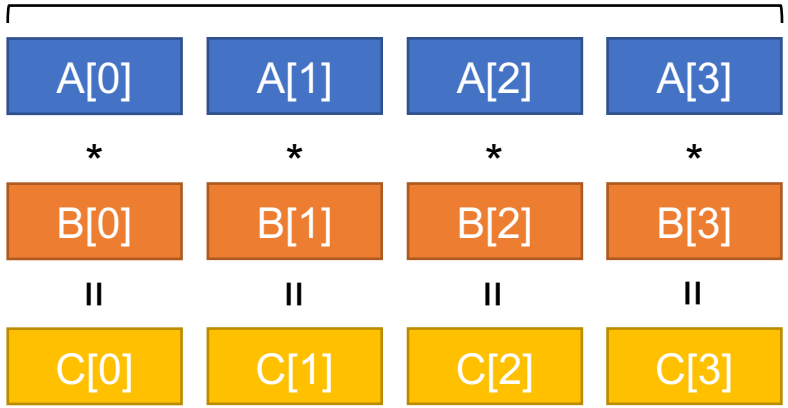
- Data array *D* much smaller than targets array *T*, and never changes
- Data array is logarithmically distributed
- Can’t use multithreading ☹
- Possibility of many search calls

Vectorization ⏩

- Compilers can produce vector (SIMD) instructions
- SIMD: Single Instruction, Multiple Data
- They do what it sounds like they do

```
for (int i = 0; i < 4; i++) {
      C[i] = A[i] * B[i];
}
```

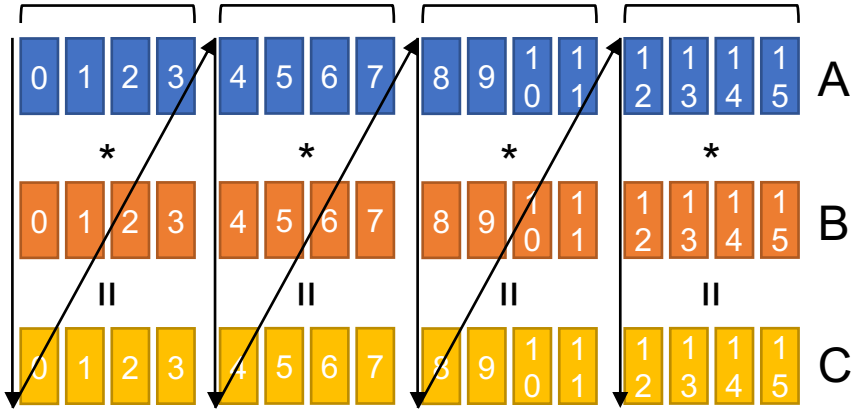
Vector width: 256 bits



Vectorization ⏩

- Compilers can produce vector (SIMD) instructions
- SIMD: Single Instruction, Multiple Data
- They do what it sounds like they do

```
for (int i = 0; i < 16; i++) {
        C[i] = A[i] * B[i];
}
```



Vectorization ⏩

- Vectorization is constrained
  - NO dependency b/w loop iterations
  - NO non-inline function calls
    - (With a few exceptions)
  - SOME branching, but it'll be *masked*: both branches are evaluated but the result from the untaken branch will be ignored
- Varies significantly by compiler

```
for (int i = 0; i < 256; i++) {
      C[i] = A[i] * B[i];
}
```

Loop can be vectorized

```
float z = 0.0f;
for (int i = 0; i < 256; i++) {
      C[i] = A[i] * B[i] + z;
      z = A[i] + B[i];
}
```

Loop can't be vectorized – memory dependency between iterations

```
for (int i = 0; i < 256; i++) {
      if (A[i] == 0) {
            ...
      } else {
            ...
      }
}
```

Loop vectorized but each iteration will be as slow as the slowest branch

Preventing Masking

- Sometimes masking can result in worse performance due to overhead
- Prevent masking by removing as much branching as possible

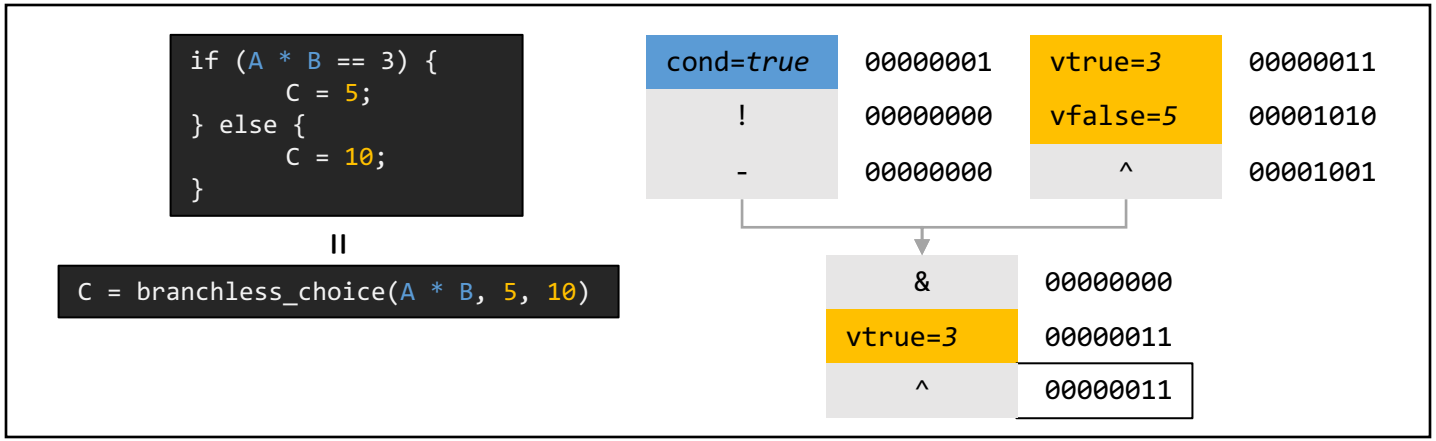
```
int branchless_choice(int cond, int vtrue, int vfalse) {
      return vtrue ^ ((vfalse ^ vtrue) & -(!cond));
}
```

Branching is simulated with bitmask

```
if (A * B == 3) {
      C = 5;
} else {
      C = 10;
}
```

=

```
C = branchless_choice(A * B, 5, 10)
```

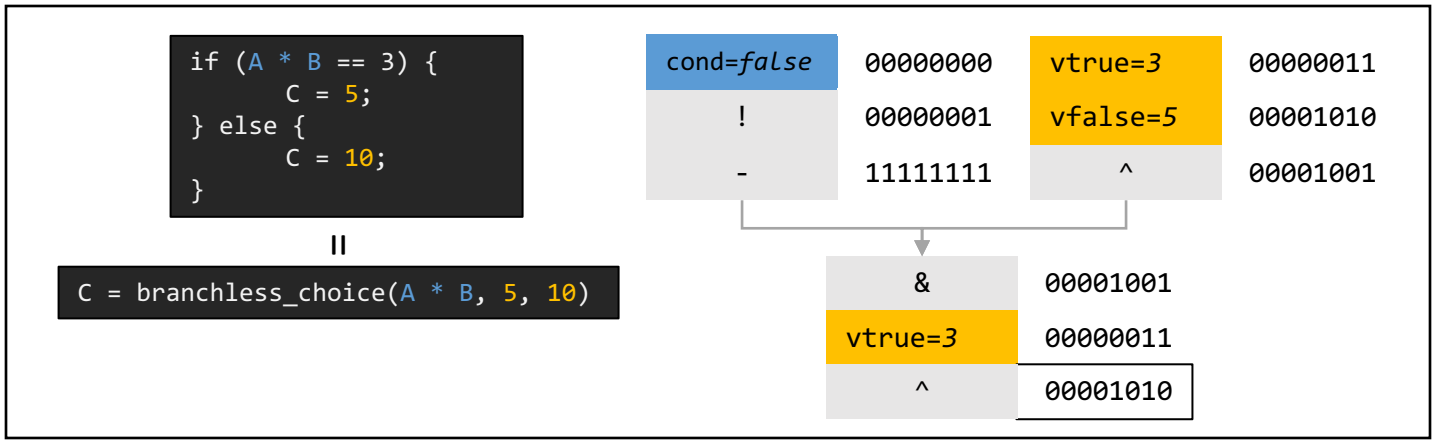


Preventing Masking

- Sometimes masking can result in worse performance due to overhead
- Prevent masking by removing as much branching as possible

```
int branchless_choice(int cond, int vtrue, int vfalse) {
      return vtrue ^ ((vfalse ^ vtrue) & -(!cond));
}
```

Branching is simulated with bitmask



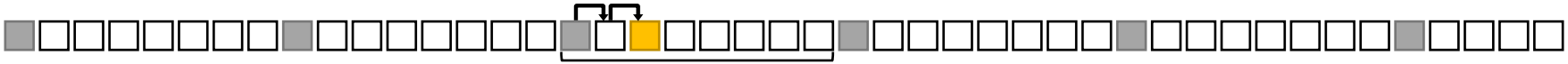
Goals 🌟

- Get us an algorithm that can do both:
  - Utilize vectorization as much as possible
  - Perform well within EOSPAC's constraints
- Data array (*D*) doesn't change! Can we use that to our advantage?

Skiplist Search ⏭️

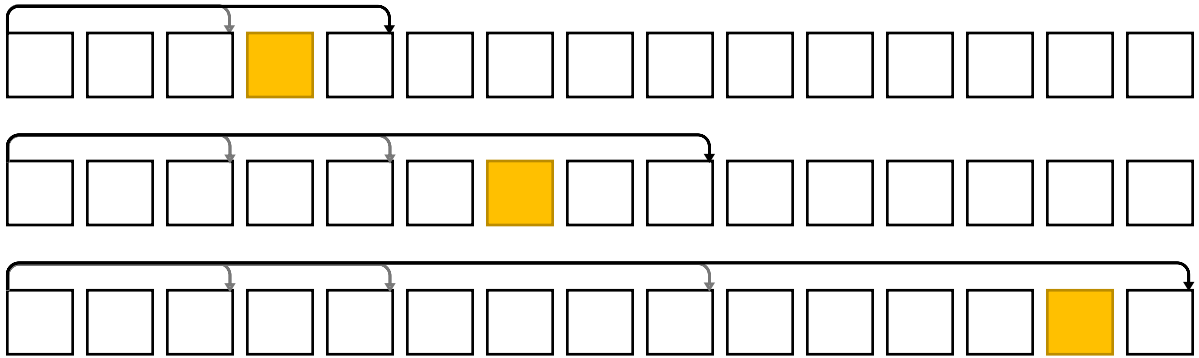
- Create “skiplist” which “skips” over 8 values (size of cache line) of *D* at a time
- Binary search on skiplist to find bounds

- Linear search through these bounds



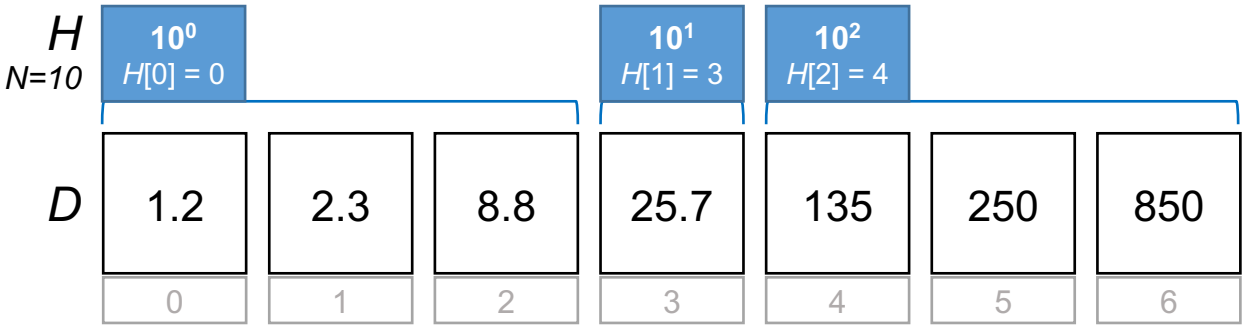
SIMD Hunt & Locate

- Modified version of Hunt & Locate with memory dependency removed
- Faster than binary search when target is near beginning of *D*



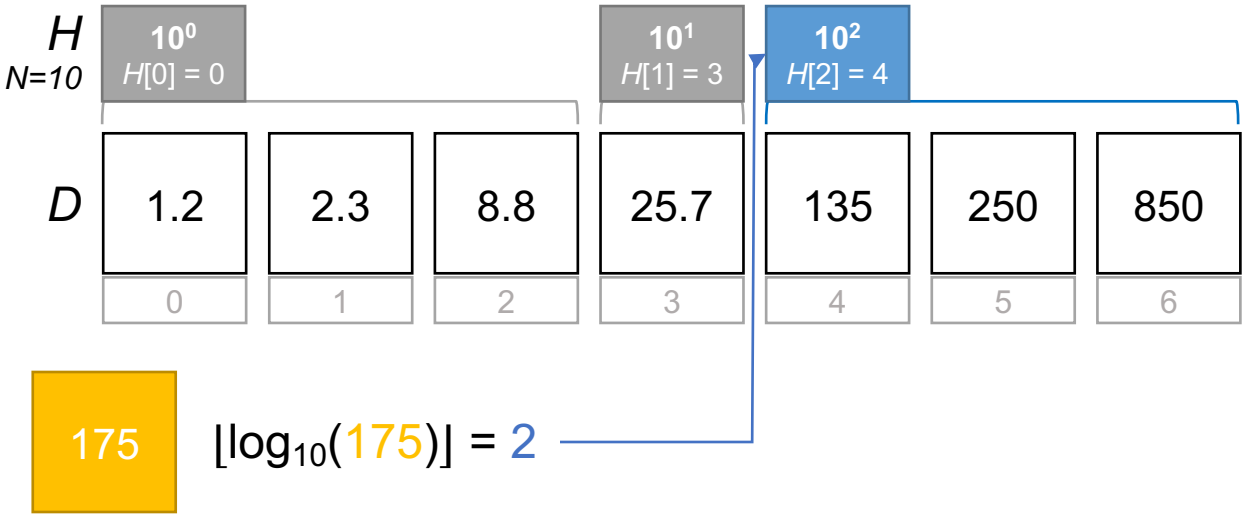
Binning-Based Search

- Bin (hash) values by the floor of their base-*N* logarithm
- Choose *N* to fit data into size of *H*, the hashtable



Binning-Based Search

- Take $\log_N$ of target, then perform secondary search in that bin
- Can use any search algo for secondary search (we tried linear and binary)



Binning-Based Search 🗑️

- Very cool! But how to choose *N*?
- Like this:

$$log_N(D_{max}) - log_N(D_{min}) = |H|$$

← We want number of powers of *N* in *D* to equal size of *H*

$$log_N\left(\frac{D_{max}}{D_{min}}\right) = |H|$$

$$\frac{D_{max}}{D_{min}} = N^{|H|}$$

$$\left(\frac{D_{max}}{D_{min}}\right)^{\frac{1}{|H|}} = N$$

- Now *N* is just the right size so that the range of values in *D* is fully contained within *H* (however large we choose to make it)

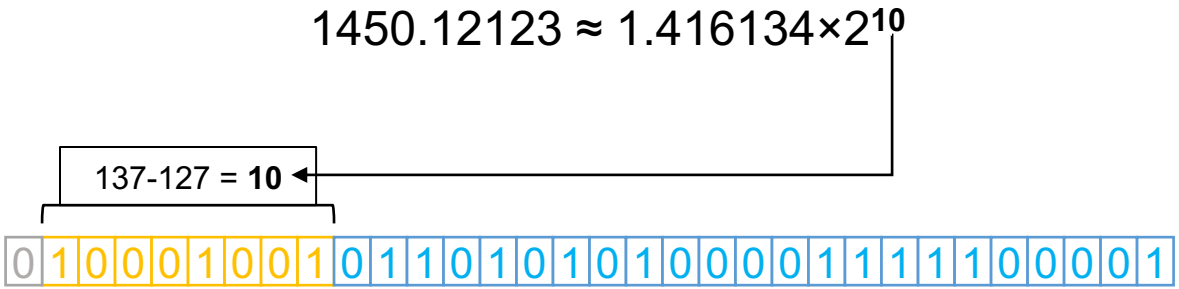
Faster

- Logarithm function is slow and just plain sucks when vectorizing 👎
- How to get log of x without `log10(x)`?

...

Faster!

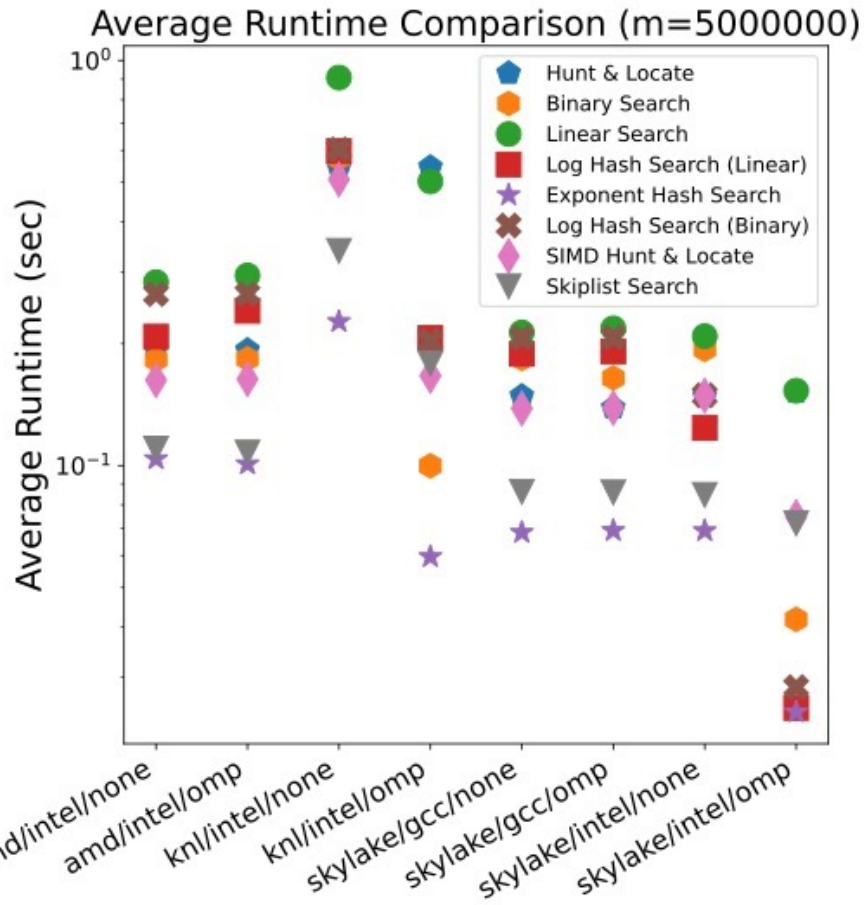
- Logarithm function is slow 👎
- How to get log of x without `log10(x)`?
- Extract exponent bits from float!



- But now the smallest *N* we can use is 2, so is it fast enough to make up for it?

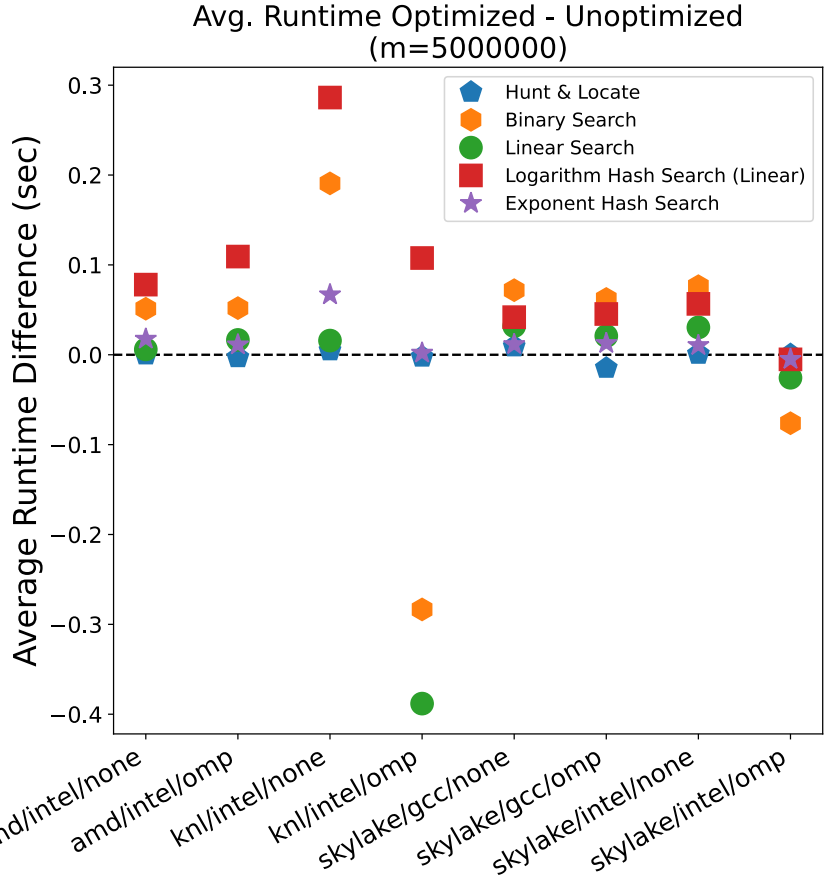
Results

- Searching for 5 mil values in size 110 array
- Only Intel compiler was able to vectorize ☹



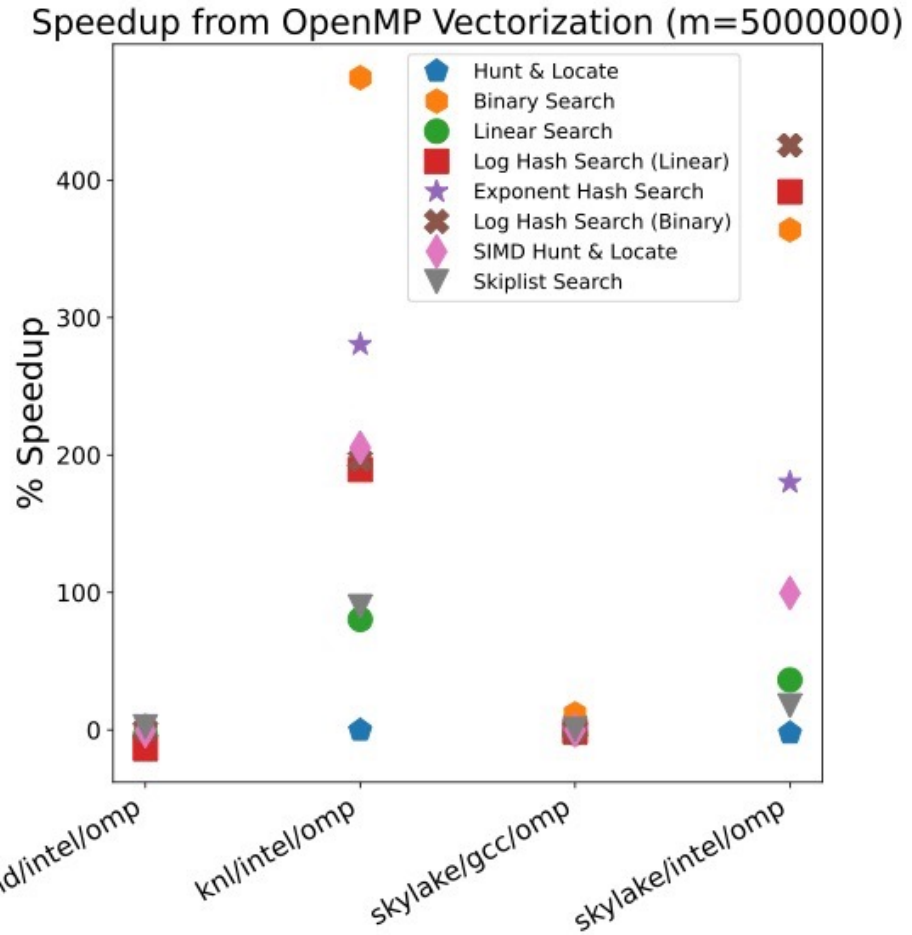
Results: Masking

- Sometimes results in worse performance
- Compare versions with and without branching



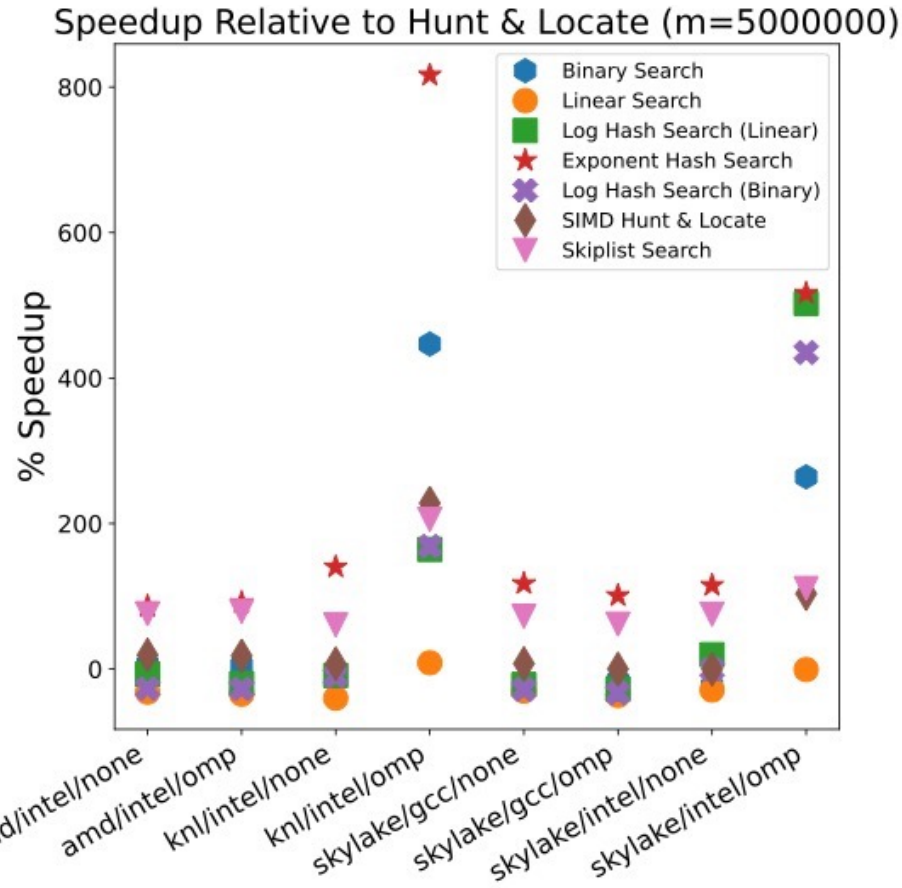
Results

- Large performance gains from vectorization



Results

- Exponent hash search does the best relative to Hunt & Locate



What Does it All Mean?

- Vectorization can be utilized to speed up certain kinds of algorithms
- Compilers are tough to wrangle
- Search time can be improved when constraints are taken into account

That’s all, folks!

Feel free to take this moment to cogitate and ask some questions